1. maven设置：配置jdk版本、仓库路径、国内下载源（Alibaba）、设置maven环境变量

<profile>

<id>jdk-1.8</id>

<activation>

<activeByDefault>true</activeByDefault>

<jdk>1.8</jdk>

</activation>

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target> <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>

</properties>

</profile>

<localRepository>D:/develop/workspaceMaven/MavenRepository</localRepository>

<mirror>

<id>alimaven</id>

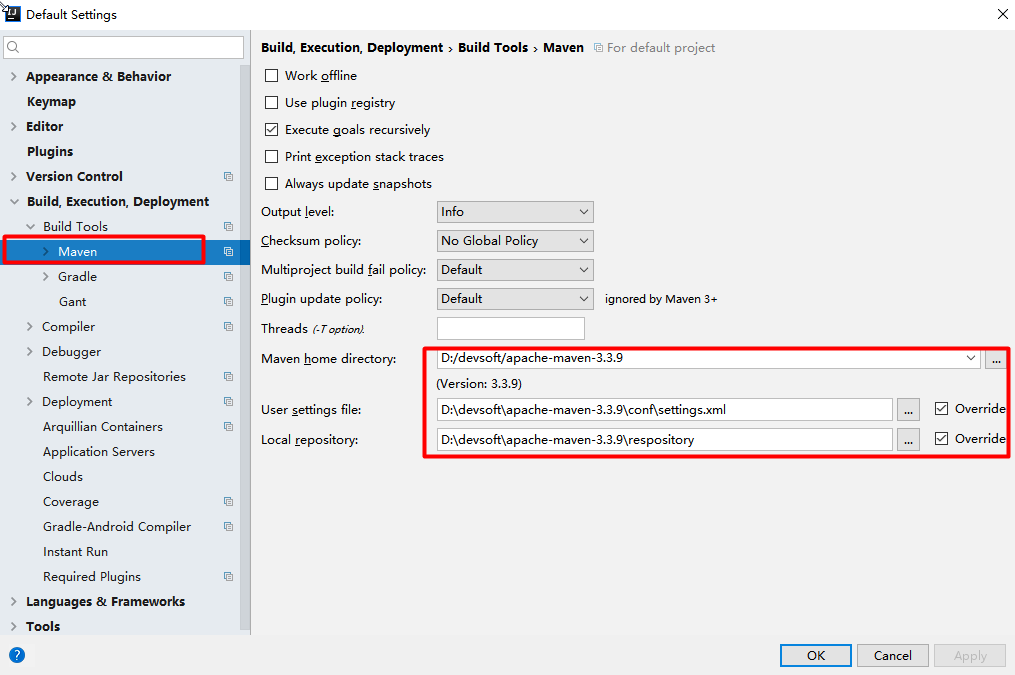
<name>aliyun maven</name>

<url>http://maven.aliyun.com/nexus/content/groups/public/</url>

<mirrorOf>central</mirrorOf>

</mirror>

2.设置IDEA的maven配置



3.使用向导创建项目，勾选必要的starters

4.在pom中添加插件可以是maven打包成一个jar，用java -jar的命令进行执行

<build>

        <plugins>

            <plugin>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring‐boot‐maven‐plugin</artifactId>

            </plugin>

        </plugins>

    </build>

5.导入配置文件处理器，配置文件进行绑定就会有提示

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring‐boot‐configuration‐processor</artifactId>

<optional>true</optional>

</dependency>

6.常用注解

@Import() 导入bean或者配置类  
@Value() 绑定bean中的值  
@Configuration 修饰配置类（spring-boot推荐+@bean）  
@ConfigurationProperties bean中的属性被指定属性文件中值绑定  
@EnableConfigurationProperties 是否启动自动文件和bean的配置

@PropertySource(value = {"classpath:per.properties"})为bean指定非全局配置文件（可以是文件组），这样不用把各种配置都放在全局配置文件中（很混乱），这样可以针对不同的bean用不同的配置实现。  
@Bean 定义返回的对象加入容器中的bean  
@Controller 前端bean  
@Service 服务端 bean  
@Repository 数据库端bean  
@Component 修饰的bean会被扫描进入容器bean  
@ComponentScan 开启扫描组件

@Validated 修饰的bean中的值需要被校验(用JSR303对属性校验)

@ImportResource(locations = {"classpath:xx.xml"})当有部分bean是用老式的xml文件配置的时候，把该注解应用到配置类上会加载该文件的bean，在locations中可以配置数组，中间逗号隔开

@Scope 确定bean范围  
@Qualifier(value = "") 如果容器中有多个类型匹配的bean，可以用此注解确定一个具体名字的bean  
@PostConstruct bean构造前调用  
@PreDestroy bean被析构时用

  @ConfigurationProperties @Value

功能 批量注入配置 一个个指定

松散绑定 支持 不支持

SpEL 不支持 支持

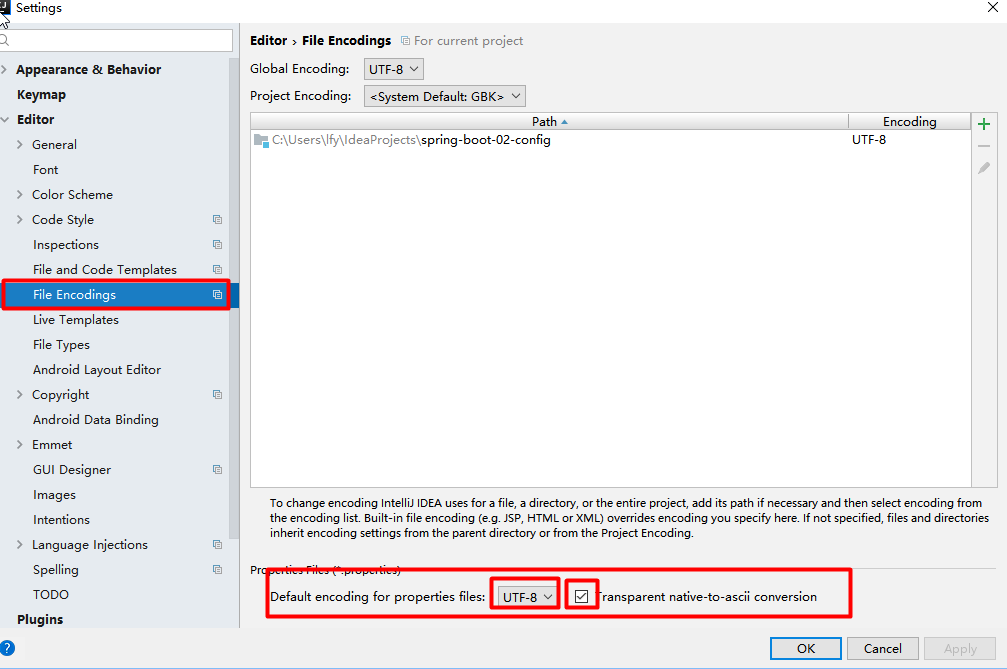
JSR303数据校验 支持 不支持

复杂类型封装 支持 不支持

如果说，我们只是在某个业务逻辑中需要获取一下配置文件中的某项值，使用@Value；而且@value用在各个bean中可以针对单个属性绑定属性文件中的某个值非常方便。

如果说，我们专门编写了一个javaBean来和配置文件进行映射，我们就直接使用@ConfigurationProperties；

7.解决属性文件中乱码的问题



8.配置文件占位符

随机数：

${random.value}、${random.int}、${random.long}

${random.int(10)}、${random.int[1024,65536]}

占位符获取之前配置的值，如果没有可以是用:指定默认值

person.last‐name=张三${random.uuid}

person.age=${random.int}

person.birth=2017/12/15

person.boss=false

person.maps.k1=v1

person.maps.k2=14

person.lists=a,b,c

person.dog.name=${person.hello:hello}\_dog

person.dog.age=15

9. 多Profile文件

主配置文件编写的时候，文件名可以是 application-{profile}.properties/yml

默认使用application.properties的配置；

激活

1、在配置文件中指定 spring.profiles.active=dev

2、命令行：

java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev；

可以直接在测试的时候，配置传入命令行参数

3、虚拟机参数；

-Dspring.profiles.active=dev

yml支持多文档块方式（这样对于properties文档方式就不需要创建那么多环境文档）

server:

  port: 8081

spring:

  profiles:

    active: prod

‐‐‐

server:

  port: 8083

spring:

  profiles: dev

‐‐‐

server:

  port: 8084

spring:

  profiles: prod  #指定属于哪个环境

10.配置文件加载位置

springboot 启动扫描以下位置application.properties或application.yml文件作为默认配置文

件

file:./config/，file:./，classpath:/config/，classpath:/

优先级由高到底，高优先级的配置会覆盖低优先级的配置；

通过spring.config.location来改变默认的配置文件位置(运维：项目打包好以后，我们可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置；指定配置文件和默认加载的这些配置文件共同起作用形成互补配置；)

Java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --spring.config.location=G:/application.properties

11.外部配置加载顺序

SpringBoot也可以从以下位置加载配置； 优先级从高到低；高优先级的配置覆盖低优先级的配置，所有的配置会

形成互补配置

命令行参数

所有的配置都可以在命令行上进行指定

java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --server.port=8087 --server.context-path=/abc

多个配置用空格分开； --配置项=值

来自java:comp/env的JNDI属性

Java系统属性（System.getProperties()）

操作系统环境变量

RandomValuePropertySource配置的random.\*属性值

由jar包外向jar包内进行寻找；

优先加载带profile

jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件

jar包内部的application-{profile}.properties或application.yml(带spring.profile)配置文件

再来加载不带profile

jar包外部的application.properties或application.yml(不带spring.profile)配置文件

jar包内部的application.properties或application.yml(不带spring.profile)配置文件

@Configuration注解类上的@PropertySource

通过SpringApplication.setDefaultProperties指定的默认属性

12.自动配置原理

SpringBoot启动的时候加载主配置类，开启了自动配置功能 @EnableAutoConfiguration

@EnableAutoConfiguration 作用：利用EnableAutoConfigurationImportSelector给容器中导入一些组件？可以查看selectImports()方法的内容；List configurations = getCandidateConfigurations(annotationMetadata, attributes);获取候选的配置

SpringFactoriesLoader.loadFactoryNames()扫描所有jar包类路径下  META‐INF/spring.factories

把扫描到的这些文件的内容包装成properties对象从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加在容器中。

每一个这样的 xxxAutoConfiguration类都是容器中的一个组件，都加入到容器中；用他们来做自动配置；每一个自动配置类进行自动配置功能；以HttpEncodingAutoConfiguration（Http编码自动配置）为例解释自动配置原理；

@Configuration   //表示这是一个配置类，以前编写的配置文件一样，也可以给容器中添加组件

@EnableConfigurationProperties(HttpEncodingProperties.class)  //启动指定类的

ConfigurationProperties功能；将配置文件中对应的值和HttpEncodingProperties绑定起来；并把

HttpEncodingProperties加入到ioc容器中

@ConditionalOnWebApplication //Spring底层@Conditional注解（Spring注解版），根据不同的条件，如果

满足指定的条件，整个配置类里面的配置就会生效；    判断当前应用是否是web应用，如果是，当前配置类生效

@ConditionalOnClass(CharacterEncodingFilter.class)  //判断当前项目有没有这个类

CharacterEncodingFilter；SpringMVC中进行乱码解决的过滤器；

@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled", matchIfMissing =

true)  //判断配置文件中是否存在某个配置  spring.http.encoding.enabled；如果不存在，判断也是成立的

//即使我们配置文件中不配置pring.http.encoding.enabled=true，也是默认生效的；

public class HttpEncodingAutoConfiguration {

   //他已经和SpringBoot的配置文件映射了

   private final HttpEncodingProperties properties;

//只有一个有参构造器的情况下，参数的值就会从容器中拿

   public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {

this.properties = properties;

}

    @Bean   //给容器中添加一个组件，这个组件的某些值需要从properties中获取

@ConditionalOnMissingBean(CharacterEncodingFilter.class) //判断容器没有这个组件？

public CharacterEncodingFilter characterEncodingFilter() {

CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();

filter.setEncoding(this.properties.getCharset().name());

filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));

filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));

return filter;

}

根据当前不同的条件判断，决定这个配置类是否生效？

一但这个配置类生效；这个配置类就会给容器中添加各种组件；这些组件的属性是从对应的properties类中获取

的，这些类里面的每一个属性又是和配置文件绑定的；

所有在配置文件中能配置的属性都是在xxxxProperties类中封装者‘；配置文件能配置什么就可以参照某个功

能对应的这个属性类

@ConfigurationProperties(prefix = "spring.http.encoding")  //从配置文件中获取指定的值和bean的属

性进行绑定

public class HttpEncodingProperties {

   public static final Charset DEFAULT\_CHARSET = Charset.forName("UTF‐8");

精髓：

1）、SpringBoot启动会加载大量的自动配置类

2）、我们看我们需要的功能有没有SpringBoot默认写好的自动配置类；

3）、我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件有，我们就不需要再来配置了）

4）、给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们就可以在配置文件中指定这

些属性的值；

13. @Conditional派生注解（Spring注解版原生的@Conditional作用）

@Conditional扩展注解 作用（判断是否满足当前指定条件）

@ConditionalOnJava 系统的java版本是否符合要求

@ConditionalOnBean 容器中存在指定Bean；

@ConditionalOnMissingBean 容器中不存在指定Bean；

@ConditionalOnExpression 满足SpEL表达式指定

@ConditionalOnClass 系统中有指定的类

@ConditionalOnMissingClass 系统中没有指定的类

@ConditionalOnSingleCandidate 容器中只有一个指定的Bean，或者这个Bean是首选Bean

@ConditionalOnProperty 系统中指定的属性是否有指定的值

@ConditionalOnResource 类路径下是否存在指定资源文件

@ConditionalOnWebApplication 当前是web环境

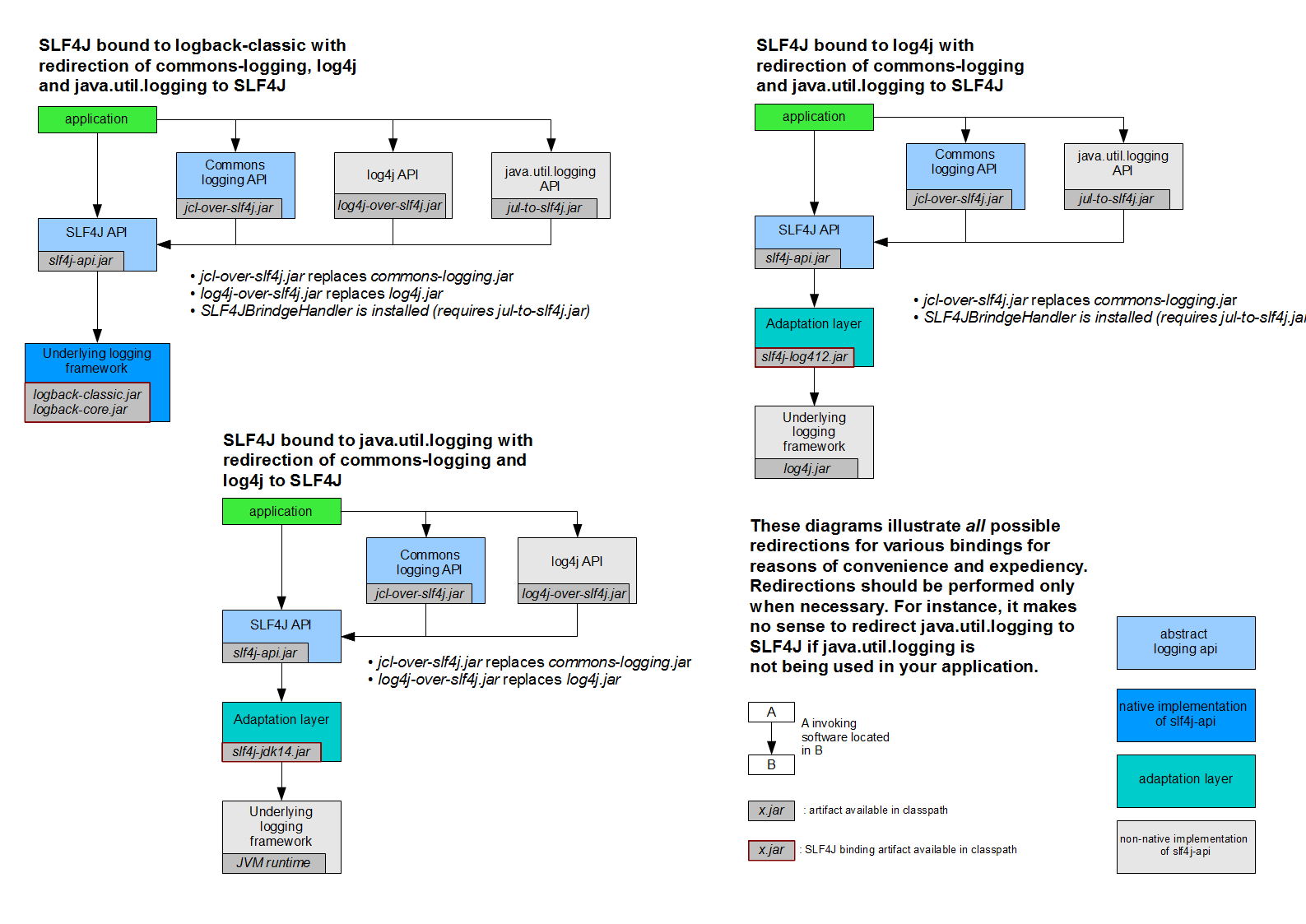
@ConditionalOnNotWebApplication 当前不是web环境

@ConditionalOnJndi JNDI存在指定项

自动配置类必须在一定的条件下才能生效；

可以通过启用 debug=true属性；来让控制台打印自动配置报告，这样我们就可以很方便的知道哪些自动配置类生效；

14. 统一日志记录，即使是别的框架和我一起统一使用slf4j进行输出？



如何让系统中所有的日志都统一到slf4j；

1、将系统中其他日志框架先排除出去；

2、用中间包来替换原有的日志框架；

3、我们导入slf4j其他的实现

15.spring-boot 的默认配置可以修改日志的基本配置

logging.level.com.atguigu=trace

#logging.path=

# 不指定路径在当前项目下生成springboot.log日志

# 可以指定完整的路径；

#logging.file=G:/springboot.log

# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹；使用 spring.log 作为默认文件

logging.path=/spring/log

#  在控制台输出的日志的格式

logging.pattern.console=%d{yyyy‐MM‐dd} [%thread] %‐5level %logger{50} ‐ %msg%n

# 指定文件中日志输出的格式

logging.pattern.file=%d{yyyy‐MM‐dd} === [%thread] === %‐5level === %logger{50} ==== %msg%n

16. 给类路径下放上每个日志框架自己的配置文件即可；SpringBoot就不使用他默认配置的了

Logback： logback-spring.xml , logback-spring.groovy , logback.xml orlogback.groovy

Log4j2： log4j2-spring.xml or log4j2.xml

JDK (Java UtilLogging)： logging.properties

logback.xml：直接就被日志框架识别了；

logback-spring.xml：日志框架就不直接加载日志的配置项，由SpringBoot解析日志配置，可以使用SpringBoot的高级Profile功能

如：

<appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">

        <!‐‐日志输出格式：%d表示日期时间， %thread表示线程名， %‐5level：级别从左显示5个字符宽度 %logger{50} 表示logger名字最长50个字符，否则按照句点分割。 %msg：日志消息， %n是换行符  ‐‐>

        <layout class="ch.qos.logback.classic.PatternLayout">

            <springProfile name="dev">

                <pattern>%d{yyyy‐MM‐dd HH:mm:ss.SSS} ‐‐> [%thread] ‐‐> %‐5level

%logger{50} ‐ %msg%n</pattern>

            </springProfile>

            <springProfile name="!dev">

                <pattern>%d{yyyy‐MM‐dd HH:mm:ss.SSS} ==== [%thread] ==== %‐5level

%logger{50} ‐ %msg%n</pattern>

            </springProfile>

        </layout>

</appender>

17.日志切换